

# Computing LZ77 in Run-Compressed Space

Alberto Policriti<sup>†\*</sup> and Nicola Prezza<sup>†</sup>

<sup>†</sup>University of Udine  
Via delle scienze, 206  
33100, Udine, Italy

alberto.policriti@uniud.it  
prezza.nicola@spes.uniud.it

<sup>\*</sup>Institute of Applied Genomics  
Via J.Linussio, 51  
33100, Udine, Italy

## Abstract

In this paper, we show that the LZ77 factorization of a text  $T \in \Sigma^n$  can be computed in  $\mathcal{O}(\mathcal{R} \log n)$  bits of working space and  $\mathcal{O}(n \log \mathcal{R})$  time,  $\mathcal{R}$  being the number of runs in the Burrows-Wheeler transform of  $T$  reversed. For extremely repetitive inputs, the working space can be as low as  $\mathcal{O}(\log n)$  bits: *exponentially* smaller than the text itself. As a direct consequence of our result, we show that a class of repetition-aware self-indexes based on a combination of run-length encoded BWT and LZ77 can be built in asymptotically optimal  $\mathcal{O}(\mathcal{R} + z)$  words of working space,  $z$  being the size of the LZ77 parsing.

## 1 Introduction

Being able to estimate the amount of self-repetitiveness of a text  $T \in \Sigma^n$  is a task that stands at the basis of many efficient compression algorithms. While fixed-order statistical methods such as empirical entropy compression are able to exploit only short text regularities [7], techniques such as Lempel-Ziv parsing (LZ77) [19], grammar compression [4], and run-length encoding of the Burrows-Wheeler transform [3, 16, 15] have been shown superior in the task of compressing highly repetitive texts. Some recent works showed moreover that such efficient representations can be augmented without asymptotically increasing their space usage in order to support also fast search functionalities [1, 5, 10] (repetition-aware self-indexes). One of the most remarkable properties of such indexes is the possibility of representing extremely repetitive texts in *exponentially* less space than that of the text itself.

Among the above mentioned repetition-aware compression techniques, LZ77 has been shown to be superior to both grammar-compression [14] and run-length encoding of the Burrows-Wheeler transform [1]. For this reason, much research is focusing into methods to efficiently build, access, and index LZ77-compressed text [2, 10]. A major concern while building LZ77-based self-indexes is to use small working space. This issue is particularly concerning in situations where the text to be parsed is extremely large and repetitive (e.g. consider the Wikipedia corpus or a large set of genomes belonging to the same species): in such cases, it is not always feasible to load the text into main memory in order to process it, even if the size of the final compressed representation could easily fit in RAM. In these domains, algorithms working in space  $\Theta(n \log n)$  [6],  $\mathcal{O}(n \log |\Sigma|)$  [12, 17], or even  $\mathcal{O}(nH_k)$  [9, 13] bits are therefore of little use as they could be exponentially more memory-demanding than the final compressed representation.

In this work, we focus on the measure of repetitiveness  $\mathcal{R}$ : the number of equal-letter runs in the BWT of the (reversed) text. Several works [1, 16, 15] studied the empirical

behavior of  $\mathcal{R}$  on highly repetitive text collections, suggesting that on such instances  $\mathcal{R}$  grows at the same rate as  $z$ . Let  $\Sigma = \{s_1, \dots, s_\sigma\}$  be the alphabet. Both  $z$  and  $\mathcal{R}$  are at least  $\sigma$  and can be  $\Theta(\sigma)$ , e.g. in the text  $(s_1 s_2 \dots s_\sigma)^e$ ,  $e > 0$ . However, the rate  $\mathcal{R}/z$  can be  $\Theta(\log_\sigma n)$ : this happens, for example, in de Bruijn sequences (of order  $k > 1$ ). In this paper, we show how to build the LZ77 parsing of a text  $T$  in space bounded by the number  $\mathcal{R}$  of runs in the BWT of  $T$  reversed. The main obstacle in computing the LZ77 parsing with a RLBWT index within *repetition-aware space* is the suffix array (SA) sampling: by sampling the SA at regular text positions, this structure requires  $\mathcal{O}((n/k) \log n)$  bits of working space and supports *locate* queries in time proportional to  $k$  (for any  $0 < k \leq n$ ). In this work we prove that—in order to compute the LZ77 parsing—it is sufficient to store at most two samples per BWT run, therefore reducing the sampling size to  $\mathcal{O}(\mathcal{R} \log n)$  bits. Our algorithm reads the text *only once* from left to right in  $\mathcal{O}(\log \mathcal{R})$  time per character (which makes it useful also in the streaming model). After reading the text,  $\mathcal{O}(n \log \mathcal{R})$  additional time is required in order to output the LZ77 phrases in text-order (the parsing itself is not stored in main memory). The total space usage is  $\mathcal{O}(\mathcal{R} \log n)$  bits.

A consequence of our result is that a class of repetition-aware self-indexes combining LZ77 and RLBWT [1] can be built in asymptotically optimal  $\mathcal{O}(z + \mathcal{R})$  words of working space. The only other known repetition-aware index that can be built in asymptotically optimal working space is based on grammar compression and is described in [18].

## 1.1 Notation

We assume that the text  $T \in \Sigma^n$ ,  $\Sigma$  being the alphabet, is terminated by a character  $\$ \in \Sigma$  not appearing elsewhere in  $T$ . The LZ77 factorization of  $T$  is defined as:

$$\mathcal{Z} = \langle pos_1, len_1, c_1 \rangle \dots \langle pos_i, len_i, c_i \rangle \dots \langle pos_z, len_z, c_z \rangle$$

where  $0 \leq pos_i, len_i < n$ ,  $c_i \in \Sigma$  for  $i = 1, \dots, z$ , and:

1.  $T = \omega_1 c_1 \dots \omega_z c_z$ , with  $\omega_i = \epsilon$  if  $len_i = 0$  and  $\omega_i = T[pos_i, \dots, pos_i + len_i - 1]$  otherwise.
2. For any  $i = 1, \dots, z$  with  $len_i > 0$ , it follows that  $pos_i < \sum_{j=1}^{i-1} (len_j + 1)$ .
3. For any  $i = 1, \dots, z$ ,  $\omega_i$  is the *longest* prefix of  $\omega_i c_i \dots \omega_z c_z$  that occurs at least twice in  $\omega_1 c_1 \dots \omega_i$ .

The notation  $\overleftarrow{S}$  indicates the reverse of the string  $S \in \Sigma^*$ . All BWT intervals are inclusive, and we denote them as pairs  $\langle l, r \rangle$  (left-right positions on the BWT). A equal-letter  $a$ -run in a string  $S$  is a substring  $W = a^e$ ,  $e > 0$  of  $S$  such that either (i)  $S = W$ , (ii)  $S = WbX$  or  $S = XbW$ ,  $b \in \Sigma$ ,  $b \neq a$ ,  $X \in \Sigma^*$ , or (iii)  $S = XbWcY$ ,  $b, c \in \Sigma$ ,  $b, c \neq a$ ,  $X, Y \in \Sigma^*$ .

A substring  $V$  of a string  $S \in \Sigma^*$  is *left-maximal* if there exist two distinct characters  $a \neq b$ ,  $a, b \in \Sigma$  such that both  $Va$  and  $Vb$  are substrings of  $S$ .

## 2 Algorithm

We now describe our algorithm, deferring a detailed description of the employed data structures to the next section. Let  $S = \#T$ ,  $\# \notin \Sigma$  being a character lexicographically smaller than all characters in  $\Sigma$ . The main structure we use is a dynamic run-length encoded BWT (RLBWT) of the text  $\overleftarrow{S}$ . Note that we index strings of the form  $\$W\#$ ,

where  $W \in (\Sigma - \{\$, \#\})^*$  and  $\$$  and  $\#$  are the LZ77 and BWT terminators, respectively. The algorithm works in two phases. In the first phase, it reads  $S$  from its first to last character, building a RLBWT representation of  $\overleftarrow{S}$ . This step employs a well-known online BWT construction algorithm which requires a dynamic string data structure to represent the BWT. The algorithm performs in total  $|S|$  *rank* and *insert* operations on the dynamic string (see [13] for a formal description of the procedure). In our case, the dynamic string is also run-length compressed (see the next section for all details).

In the second phase, the algorithm scans  $S$  from left to right by using the BWT just built (i.e. by repeatedly applying LF mapping starting from character  $\#$ ) and outputs the LZ77 factors. We enumerate text positions in  $S$ -order (despite the fact that we are indexing  $\overleftarrow{S}$ ). Since characters in  $S = \#T$  are right-shifted by 1 position with respect to  $T$ , we enumerate  $S$ -positions starting from  $-1$ , so that  $T[i] = S[i]$ ,  $0 \leq i < n$  (this simplifies notation).

While reading the  $j$ -th character of  $S$ ,  $j \geq 0$ , we search in the index the current (reversed) LZ phrase prefix  $\overleftarrow{S}[i, \dots, j]$ ,  $j \geq i \geq 0$  ( $i$  being the first position of the current phrase). If none of the positions in the BWT interval associated with  $\overleftarrow{S}[i, \dots, j]$  correspond to an occurrence of  $S[i, \dots, j]$  in  $S[0, \dots, j-1]$ , then we output the LZ triple  $\langle t, j-i, S[j] \rangle$ , where  $t < i$  and  $S[t, \dots, t+j-i-1] = S[i, \dots, j-1]$  (if  $i = j$ , then  $t = \text{NULL}$ ). The problems to solve are (i) determine whether or not the BWT interval contains occurrences of the phrase prefix in  $S[0, \dots, j-1]$ , and—if the answer is negative and  $j > i$ —(ii) find  $t < i$  such that  $S[t, \dots, t+j-i-1] = S[i, \dots, j-1]$ . We now show how to answer these queries in  $\mathcal{O}(\mathcal{R} \log n)$  bits of working space by keeping in memory  $\sigma$  dynamic sets containing in total  $\mathcal{O}(\mathcal{R})$  SA samples.

## 2.1 Suffix Array Sampling

From now on, we will write  $BWT$  to indicate the Burrows-Wheeler transform of the *whole* string  $\overleftarrow{S}$ . Note that, even though we say we *sample the suffix array*, we actually sample text positions associated with BWT positions (i.e. we sample  $S$ -positions on the  $L$  column instead of  $S$ -positions on the  $F$  column of the BWT matrix). Moreover, since we enumerate positions in  $S$ -order (not  $\overleftarrow{S}$ -order), BWT position  $k$  will be associated with the sample  $n - SA[k]$ ,  $SA[k]$  being the  $k$ -th entry in the suffix array of  $\overleftarrow{S}$ .

Let  $0 \leq j < n$  be a  $S$ -position, and  $0 \leq k < n+1$  be its corresponding position on the BWT. We store SA samples as pairs  $\langle j, k \rangle$ . Each pair is of one of three types: *singleton*, denoted as  $\langle j, k \rangle^\circ$ , *open*, denoted as  $\lceil \langle j, k \rangle$ , and *close*, denoted as  $\langle j, k \rceil$ . If the pair type is not relevant for the discussion, we simply write  $\langle j, k \rangle$ .

Let  $\Sigma = \{s_1, \dots, s_\sigma\}$  be the alphabet. The samples are stored in  $\sigma$  red-black trees  $\mathcal{B}_{s_1}, \dots, \mathcal{B}_{s_\sigma}$ , and are ordered by BWT coordinate (i.e. the second component of the pairs). While reading character  $a = S[j] = BWT[k]$ , we first locate the inclusive bounds  $l \leq k \leq r$  of its associated BWT  $a$ -run. We update the trees according to the following rules:

1. If no pair  $\langle j', k' \rangle \in \mathcal{B}_a$  is such that  $l \leq k' \leq r$ , then we insert the singleton  $\langle j, k \rangle^\circ$  in  $\mathcal{B}_a$ .
2. If there exists a singleton pair  $\langle j', k' \rangle^\circ \in \mathcal{B}_a$  such that  $l \leq k' \leq r$ , then we remove it and:
  - (a) If  $k < k'$ , then we insert in  $\mathcal{B}_a$  the pairs  $\lceil \langle j, k \rangle$  and  $\langle j', k' \rceil$
  - (b) If  $k' < k$ , then we insert in  $\mathcal{B}_a$  the pairs  $\lceil \langle j', k' \rangle$  and  $\langle j, k \rceil$

3. If there exist two pairs  $[\langle j', k' \rangle, \langle j'', k'' \rangle] \in \mathcal{B}_a$  such that  $l \leq k' < k'' \leq r$ :

- (a) If  $k < k'$ , then we remove  $[\langle j', k' \rangle]$  from  $\mathcal{B}_a$  and insert  $[\langle j, k \rangle]$  in  $\mathcal{B}_a$
- (b) If  $k > k''$ , then we remove  $[\langle j'', k'' \rangle]$  from  $\mathcal{B}_a$  and insert  $[\langle j, k \rangle]$  in  $\mathcal{B}_a$
- (c) Otherwise ( $k' < k < k''$ ), we leave the trees unchanged.

We say that a BWT  $a$ -run  $BWT[l, \dots, r]$  *contains a pair* or, equivalently, *contains a SA sample* if there exists some  $\langle j, k \rangle \in \mathcal{B}_a$  such that  $l \leq k \leq r$ . Moreover, we say that BWT-position  $k$  is *marked with a SA sample* if  $\langle j, k \rangle \in \mathcal{B}_a$ , where  $a = S[j] = BWT[k]$  ( $j$  is the  $S$ -position corresponding to BWT-position  $k$ ).

It is easy to see that the following invariants hold after the application of any of the above three rules: (i) each BWT run contains either no pairs, a singleton pair, or two pairs—one open and one close; (ii) If a BWT run contains an open  $[\langle j', k' \rangle]$  and a close  $[\langle j'', k'' \rangle]$  pair, then  $k' < k''$ ; (iii) once we add a SA sample inside a BWT run, that run will—from that moment on—always contain at least one SA sample.

By saying that we have *processed  $S$ -positions*  $0, \dots, j$ , we mean that—starting with all trees empty—we have applied the update rules to the SA samples  $\langle 0, 0 \rangle, \langle 1, BWT.LF(0) \rangle, \langle 2, BWT.LF^2(0) \rangle, \dots, \langle j, BWT.LF^j(0) \rangle$ , where  $BWT.LF^i(0)$  denotes the LF function applied  $i$  times to the BWT-position 0 (e.g.  $BWT.LF^2(0) = BWT.LF(BWT.LF(0))$ ).

We now prove that, after processing  $S$ -positions  $0, \dots, j$ , we can locate at least one occurrence of any string that occurs in  $S[0, \dots, j]$ . This property will allow us to locate LZ phrase boundaries and previous occurrences of LZ phrases.

Suppose we have processed  $S$ -positions  $0, \dots, j$ , and let  $[l, r]$  be the BWT interval associated with a *left-maximal* string  $\overleftarrow{V} \in \Sigma^*$ . The following holds:

**Lemma 1** *There exists a pair  $\langle j', k' \rangle \in \mathcal{B}_a$  such that  $l \leq k' \leq r$  if and only if  $Va$  occurs in  $S[0, \dots, j]$ .*

**Proof 1** ( $\Rightarrow$ ) *If such a pair  $\langle j', k' \rangle \in \mathcal{B}_a$  exists, where  $l \leq k' \leq r$ , then clearly  $S[j' - m, \dots, j'] = Va$ . Moreover, since we processed only  $S$ -positions  $0, \dots, j$ , it holds that  $j' \leq j$ , therefore  $Va$  occurs in  $S[0, \dots, j]$ .*

( $\Leftarrow$ ) *Let  $S[t, \dots, t + m] = Va$ , with  $t \leq j - m$ . One of the following 2 cases can happen:*

(1) *The BWT  $a$ -run containing character  $S[t + m] = a$  is a substring of  $BWT[l, \dots, r]$  and is neither a prefix nor a suffix of  $BWT[l, \dots, r] = Xca^e dY$ , for some  $X, Y \in \Sigma^*$ ,  $c, d \neq a, e > 0$ . Then—for invariant (iii) and rule 1—since we have visited it (while processing  $S$ -position  $t + m$ ), the  $a$ -run must contain at least one SA sample.*

(2) *The BWT  $a$ -run containing character  $S[t + m] = a$  spans at least two BWT intervals ( $\langle l, r \rangle$  included) or is a suffix/prefix of  $BWT[l, \dots, r]$ . Since  $V$  is left-maximal in  $S$ , then  $BWT[l, \dots, r]$  contains also a character  $b \neq a$ . We therefore have that either (i)  $BWT[l, \dots, r] = a^e XbY$ , or (ii)  $BWT[l, \dots, r] = YbXa^e$ , where  $X, Y \in \Sigma^*$ ,  $e > 0$ . The two cases are symmetric, so here we prove only (i).*

*Consider all  $\overleftarrow{S}$ -suffixes  $\overleftarrow{S}[0, \dots, j'']$  such that*

- $j'' \leq j$
- $Va$  is a suffix of  $S[0, \dots, j'']$
- *The lexicographic rank of  $\overleftarrow{S}[0, \dots, j'' - 1]$  among all  $\overleftarrow{S}$ -suffixes is  $l \leq k'' \leq l + e - 1$  (i.e. the suffix lies in  $BWT[l, \dots, l + e - 1] = a^e$ ).*

There exists at least one such  $\overleftarrow{S}$ -suffix:  $\overleftarrow{S}[0, \dots, t + m]$ . Then, it is easy to see that the rank  $k'$  of the lexicographically largest<sup>1</sup>  $\overleftarrow{S}$ -suffix with the above properties is such that  $\langle j', k' \rangle \in \mathcal{B}_a$  for some  $j' \leq j$ . This is implied by the three update rules described above. The BWT position  $k$  corresponding to  $S$ -position  $t + m$  lies in the BWT interval  $[l, l + e - 1]$ , therefore either (i)  $k$  is the rightmost position visited in its run (thus it is marked with a SA sample), or (ii) the rightmost visited position  $k' > k$  in  $[l, l + e - 1]$  is marked with a SA sample (note that lexicographically largest translates to rightmost on BWT intervals).

As a corollary, we note that we can drop the left-maximality requirement from Lemma 1. Suppose we have processed  $S$ -positions  $0, \dots, j - 1$  (none if  $j = 0$ ). The following holds:

**Corollary 1** *After processing  $S$ -positions  $j, \dots, j + m - 1$ ,  $m > 0$ , if a string  $W \in \Sigma^m$  occurs in  $S[0, \dots, j + m - 1]$ , then we can locate one of such occurrences.*

**Proof 2** *We prove the property by induction on  $|W| = m > 0$ . Let  $W = Va$ ,  $V \in \Sigma^*$ ,  $a \in \Sigma$ . If  $m = 1$ , then  $V = \epsilon$  (empty string), and the BWT interval associated with  $\overleftarrow{V}$  is the full interval  $\langle 0, n \rangle$ . But then,  $\text{BWT}[0, \dots, n]$  contains at least 2 distinct characters ( $a$  and  $\#$ ), so we can apply Lemma 1 to find a previous occurrence of  $W = a$ .*

*If  $m > 1$ , then  $|V| > 0$  and two cases can occur. If the BWT interval associated with  $\overleftarrow{V}$  contains at least 2 distinct characters ( $a$  included since by hypothesis  $Va$  occurs in  $S[0, \dots, j + m - 1]$ ), then we can apply Lemma 1 to find an occurrence of  $W = Va$  in  $S[0, \dots, j + m - 1]$ . If, on the other hand, the BWT interval associated with  $\overleftarrow{V}$  contains only one distinct character, then this character must be  $a$  since  $Va$  occurs in  $S$ . By inductive hypothesis we can locate an occurrence  $\text{occ}$  of  $V$  in  $S[0, \dots, j + m - 2]$ . But then, since all occurrences of  $V$  in  $S$  are followed by  $a$ ,  $\text{occ}$  is also an occurrence of  $W = Va$  in  $S[0, \dots, j + m - 1]$ .*

## 2.2 Pseudocode

Our complete procedure is reported as Algorithm 1. In line 1 we build the RLBWT of  $\overleftarrow{S} = \overleftarrow{\#T}$  using the online algorithm mentioned at the beginning of this section and employing a dynamic run-length encoded string data structure to represent the BWT. This is the only step that uses the input text, which is read only once from left to right. Since the dynamic string we use is run-length compressed, this step requires  $\mathcal{O}(\mathcal{R} \log n)$  bits of working space.

From lines 2 to 9 we initialize all variables that will be used during the algorithm execution. In order: the text length  $n$ , the current position  $j$  on  $T$ , the position  $k$  on  $RLBWT$  corresponding to position  $j$  on  $T$ , the current LZ77 phrase prefix length  $\text{len}$  (last character  $T[j]$  excluded), the  $T$ -position  $\text{occ} < j$  at which the current phrase prefix  $T[j - \text{len}, \dots, j - 1]$  occurs ( $\text{occ} = \text{NULL}$  if  $\text{len} = 0$ ), the red-black trees  $\mathcal{B}_{s_1}, \dots, \mathcal{B}_{s_\sigma}$  used to store the SA samples, the current character  $c = T[j] = RLBWT[k]$  on the text, and the (inclusive) interval  $\langle l, r \rangle$  corresponding to the current reversed LZ phrase prefix  $\overleftarrow{T[j - \text{len}, \dots, j - 1]}$  on  $RLBWT$  ( $\langle l, r \rangle$  is the full interval  $\langle 0, n \rangle$  if  $\text{len} = 0$ ).

The *while* cycle in line 10 scans  $T$  positions from the first to last. First of all, we have to discover if the current character  $T[j] = c$  ends a LZ phrase. In line 11 we count the number  $u$  of runs that intersect interval  $[l, r]$  on  $RLBWT$ . If  $u = 1$ , then the current phrase prefix  $T[j - \text{len}, \dots, j - 1]$  is always followed by  $c$  in  $T$ , and consequently  $T[j]$  cannot be the last character of the current LZ phrase. Otherwise, by Lemma 1  $T[j - \text{len}, \dots, j]$  occurs in

<sup>1</sup>To prove the symmetric case (ii), use the lexicographically *smallest* suffix of this kind.

$T[0, \dots, j-1]$  if and only if there exists a SA sample  $\langle j', k' \rangle \in \mathcal{B}_c$  such that  $l \leq k' \leq r$ . The existence of such pair can be verified with a binary search on the red-black tree  $\mathcal{B}_c$ . In line 12 we perform these two tests. If at least one of these two conditions holds, then  $T[j-len, \dots, j]$  occurs in  $T[0, \dots, j-1]$  and therefore is not a LZ phrase. If this is the case, we now have to find  $occ < (j-len)$  such that  $T[occ, \dots, occ+len] = T[j-len, \dots, j]$  (i.e. a previous occurrence of the current LZ phrase prefix). This goal can be achieved by applying the inductive proof of Corollary 1. If  $u = 1$  then  $occ$  is already the value we need. Otherwise (Lines 13-14) we find a SA sample  $\langle j', k' \rangle \in \mathcal{B}_c$  such that  $l \leq k' \leq r$  (it must exist since  $u > 1$  and the condition in Line 12 succeeded). Procedure  $\mathcal{B}_c.locate(l, r)$  returns such  $j'$  (to make the procedure deterministic, one could return the value  $j'$  associated with the smallest BWT position  $l \leq k' \leq r$ ). Then, we assign to  $occ$  the value  $j' - len$  (Line 14). We can now increase the current LZ phrase prefix length (Line 15) and update the BWT interval  $\langle l, r \rangle$  so that it corresponds to the string  $\overleftarrow{T[j-len+1, \dots, j]}$  (LF mapping in Line 16).

If both the conditions at line 12 fail, then the string  $T[j-len, \dots, j]$  does not occur in  $T[0, \dots, j-1]$  and therefore is a LZ phrase. By the inductive hypothesis of Corollary 1,  $occ < j-len$  is either *NULL*—if  $len = 0$ —or such that  $T[occ, \dots, occ+len-1] = T[j-len, \dots, j-1]$  otherwise. At line 18 we can therefore output the LZ factor. We now have to open (and start searching in RLBWT) a new LZ phrase: at lines 19-21 we reset the current phrase prefix length, set  $occ$  to *NULL*, and reset the interval associated to the current (reversed) phrase prefix to the full interval.

All we have left to do now is to process position  $j$  (i.e. apply the update rules to the SA sample  $\langle j, k \rangle$ ) and proceed to the next text position. At line 22 we locate the (inclusive) borders  $\langle l_{run}, r_{run} \rangle$  of the BWT run containing position  $k$  (i.e.  $l_{run} \leq k \leq r_{run}$ ). This information is used at line 23 to apply the update rules on  $\mathcal{B}_c$  and on the SA sample  $\langle j, k \rangle$ . Finally, we increase the current  $T$ -position  $j$  (line 24), compute the corresponding position  $k$  on RLBWT (line 25), and read the next  $T$ -character  $c$  on the RLBWT.

### 3 Structures

To implement the RLBWT data structure, we adopt the approach of [16] (RLFM+ index): we store one character per run in a string  $H \in (\Sigma \cup \{\#\})^{\mathcal{R}}$ , we mark the beginning of the runs with a bitvector  $V_{all}[0, \dots, n]$ , and we store the lengths of  $c$ -runs in  $\sigma$  bitvectors  $V_c$ ,  $c \in \Sigma$  (i.e. length  $m$  is encoded as  $10^{m-1}$ )<sup>2</sup>. In this way, *rank-select-access* operations on the BWT are reduced to *rank-select-access* operations on  $H$  and on the bitvectors. If the bitvectors are gap-encoded, the structure takes  $\mathcal{O}(\mathcal{R} \log n)$  bits of space.

The above representation can support also character insertions: an insertion in the run-length string can be easily implemented with a constant number of insertions in  $H$  and insertions and 0-deletions—i.e. deleting a 0-bit—in the bitvectors. For space constraints, we do not give these details here.

In order to support insertions, all structures must be dynamic. For  $H$  we can use the structure of [11] ( $\mathcal{O}(\mathcal{R} \log n)$  bits of space and  $\mathcal{O}(\log \mathcal{R})$ -time rank, select, access, and insert). We encode the bitvectors with partial sums data structures. Since we want these structures to be dynamic, we are interested in the *Searchable Partial Sums with Indels* (SPSI) problem. This problem consists in maintaining a sequence  $s_1, \dots, s_m$  of nonnegative

<sup>2</sup>For example, let  $BWT = bc\#bbbccccbaaaaaaaa$ . Then,  $H = bc\#bcba$ ,  $V_{all} = 111100010001100000000000$ ,  $V_a = 10000000000$ ,  $V_b = 110001$ , and  $V_c = 11000$  ( $V_{\#}$  is always 1).

---

**Algorithm 1:** LZ77\_in\_RLE\_space( $T$ )

---

**input** : A text  $T \in \Sigma^n$  ending with  $\$ \in \Sigma$

**output:** LZ77 factors of  $T$  in text order.

```
1  $RLBWT \leftarrow build\_rev\_RLBWT(\#T);$       /* Build online the BWT of  $\overleftarrow{\#T}$  */
2  $n \leftarrow |T|;$                           /*  $T$  length */
3  $j \leftarrow 0;$                             /* Last position (on  $T$ ) of current LZ phrase prefix */
4  $k \leftarrow 0;$                             /* Position on  $RLBWT$  corresponding to  $T[j]$  */
5  $len \leftarrow 0;$                           /* Length of current LZ phrase prefix */
6  $occ \leftarrow NULL;$                       /* Previous occurrence of current LZ phrase prefix */
7  $\mathcal{B}_{s_1}, \dots, \mathcal{B}_{s_\sigma} \leftarrow \emptyset;$  /* Initialize red-black trees of SA samples */
8  $c \leftarrow RLBWT[k];$                       /* Current  $T$  character */
9  $\langle l, r \rangle \leftarrow \langle 0, n \rangle;$         /* Range of current LZ phrase prefix in  $RLBWT$  */
10 while  $j < n$  do
11    $u \leftarrow RLBWT.number\_of\_runs(l, r);$  /* Runs intersecting  $\langle l, r \rangle$  */
12   if  $u = 1$  or  $\mathcal{B}_c.exists\_sample(l, r)$  then
13     if  $u > 1$  then
14        $occ \leftarrow \mathcal{B}_c.locate(l, r) - len;$  /* Occurrence of phrase prefix */
15        $len \leftarrow len + 1;$                 /* Increase length of current LZ phrase */
16        $\langle l, r \rangle \leftarrow RLBWT.LF(\langle l, r \rangle, c);$  /* Backward search step */
17   else
18     Output  $\langle occ, len, c \rangle;$                 /* Output LZ77 factor */
19      $len \leftarrow 0;$                         /* Reset phrase prefix length */
20      $occ \leftarrow NULL;$                     /* Reset phrase prefix occurrence */
21      $\langle l, r \rangle \leftarrow \langle 0, n \rangle;$       /* Reset range of current LZ phrase prefix */
22    $\langle l_{run}, r_{run} \rangle \leftarrow RLBWT.locate\_run(k);$  /* run of BWT position  $k$  */
23    $\mathcal{B}_c.update\_tree(\langle j, k \rangle, \langle l_{run}, r_{run} \rangle);$  /* Apply update rules */
24    $j \leftarrow j + 1;$                         /* Increment  $T$  position */
25    $k \leftarrow RLBWT.LF(k);$                   /* RLBWT position corresponding to  $j$  */
26    $c \leftarrow RLBWT[k];$                     /* Read next  $T$  character */
```

---

integers, each one of  $k \in \mathcal{O}(w)$  bits,  $w$  being the size of a memory word. We want to support the following operations in  $\mathcal{O}(\log m)$  time with a structure  $PS$  of  $\mathcal{O}(m \log M)$  bits, where  $M = \sum_{i=1}^m s_i$ :

- $PS.sum(i) = \sum_{j=1}^i s_j$
- $PS.search(x)$  is the smallest  $i$  such that  $\sum_{j=0}^i \geq x$
- $PS.update(i, \delta)$ : update  $s_i$  to  $s_i + \delta$ .  $\delta$  can be negative as long as  $s_i + \delta \geq 0$ .
- $PS.insert(i)$ : insert 0 between  $s_{i-1}$  and  $s_i$  (if  $i = 0$ , 0 becomes the first element).

We do not need *delete* since we update the bitvectors only with insertions. Once having such a structure, a length- $n$  bitvector  $B = 10^{s_1-1}10^{s_2-1}...10^{s_m-1}$  ( $s_i > 0$ ) can be encoded in  $\mathcal{O}(m \log n)$  bits of space with a partial sum  $PS$  on the sequence  $s_1, \dots, s_m$ . We need to answer the following queries on  $B$ :  $B[i]$  (*access*),  $B.rank(i) = \sum_{j=0}^i B[j]$ ,  $B.select(i)$  (the position  $j$  such that  $B[j] = 1$  and  $B.rank(j) = i$ ),  $B.insert(i, b)$  (insert bit  $b \in \{0, 1\}$  between positions  $i - 1$  and  $i$ ), and  $B.delete_0(i)$ , where  $B[i] = 0$  (delete  $B[i]$ ).

It is easy to see that *rank/access* and *select* operations on  $B$  reduce to *search* and *sum* operations on  $PS$ , respectively.  $B.delete_0(i)$  requires just a search and an update on  $PS$ . To support *insert* on  $B$ , we can operate as follows.  $B.insert(i, 0)$ ,  $i > 0$ , is implemented with  $PS.update(PS.search(i), 1)$ .  $B.insert(0, 1)$  is implemented with  $PS.insert(0)$  followed by  $PS.update(0, 1)$ .  $B.insert(i, 1)$ ,  $i > 0$ , “splits” an integer into two integers: let  $j = PS.search(i)$  and  $\delta = PS.sum(j) - i$ . We first decrease  $s_j$  with  $PS.update(j, -\delta)$ . Then, we insert a new integer  $\delta + 1$  with the operations  $PS.insert(j + 1)$  and  $PS.update(j + 1, \delta + 1)$ .

### 3.1 SPSI implementation

In our case, the bit length of each integer in the partial sums structures is  $k = \log n$ ,  $n$  being the text length. The total number of integers to be stored among the  $\sigma + 1$  partial sums is  $2\mathcal{R}$ . Since we aim at obtaining  $\mathcal{O}(\mathcal{R} \log n)$  bits of space, the problem can be solved by using red-black trees (see [8] for a similar construction).

Let  $s_1, \dots, s_m$  be a sequence of nonnegative integers. We store  $s_1, \dots, s_m$  in the leaves of a red-black tree, and we store in each internal node of the tree the number of nodes and partial sum of its subtrees. *Sum* and *search* queries can then be easily implemented with a traversal of the tree from the root to the target leaf. *Update* queries require finding the integer (leaf) of interest and then updating  $\mathcal{O}(\log m)$  partial sums while climbing the tree from the leaf to the root. Finally, *insert* queries require finding an integer (leaf)  $s_i$  immediately preceding or following the insert position, substituting it with an internal node with two children leaves  $s_i$  and 0 (the order depending on the insert position—before or after  $s_i$ ), adding 1 to  $\mathcal{O}(\log m)$  subtree-size counters while climbing the tree up to the root, and applying the RBT update rules. This last step requires the modification of  $\mathcal{O}(1)$  counters (subtree-size/partial sum) if RBT rotations are involved. All operations take  $\mathcal{O}(\log m)$  time.

### 3.2 Analysis

It is easy to see that *rank*, *select*, *access*, and *insert* operations on RLBWT take  $\mathcal{O}(\log \mathcal{R})$  time each. Operations  $\mathcal{B}_c.exists\_sample(l, r)$  (line 12) and  $\mathcal{B}_c.locate(l, r)$  (Line 14) require just a binary search on the red-black tree of interest and can also be implemented in



$\mathcal{O}(\log \mathcal{R})$  time.  $RLBWT.number\_of\_runs(l, r)$  is the number of bits set in  $V_{all}[l, \dots, r]$ , plus 1 if  $V_{all}[l] = 0$ : this operation requires therefore  $\mathcal{O}(1)$  rank/access operations on  $V_{all}$  ( $\mathcal{O}(\log \mathcal{R})$  time). Similarly,  $RLBWT.locate\_run(k)$  requires finding the two bits set preceding and following position  $k$  in  $V_{all}$  ( $\mathcal{O}(\log \mathcal{R})$  time with a constant number of rank and select operations). We obtain:

**Theorem 1** *Algorithm 1 computes the LZ77 factorization of a text  $T \in \Sigma^n$  in  $\mathcal{O}(\mathcal{R} \log n)$  bits of working space and  $\mathcal{O}(n \log \mathcal{R})$  time,  $\mathcal{R}$  being the number of runs in the Burrows-Wheeler transform of  $T$  reversed.*

As a direct result of Theorem 1, we obtain an asymptotically optimal-space construction algorithm for a class of repetition-aware indexes [1] combining a RLBWT with the LZ77 factorization. The construction of such indexes requires building the RLBWT, computing the LZ77 factorization of  $T$ , and building additional structures of  $\mathcal{O}(z)$  words of space. We observe that with our algorithm all these steps can be easily carried out in  $\mathcal{O}(\mathcal{R} + z)$  words of working space, which is asymptotically optimal.

## References

- [1] Djamel Belazzougui, Fabio Cunial, Travis Gagie, Nicola Prezza, and Mathieu Raffinot. Composite repetition-aware data structures. In *Proc. CPM*, pages 26–39, 2015.
- [2] Djamel Belazzougui, Travis Gagie, Paweł Gawrychowski, Juha Kärkkäinen, Alberto Ordóñez, Simon J Puglisi, and Yasuo Tabei. Queries on lz-bounded encodings. *arXiv preprint arXiv:1412.0967*, 2014.
- [3] Michael Burrows and David J Wheeler. A block-sorting lossless data compression algorithm. 1994.
- [4] Moses Charikar, Eric Lehman, Ding Liu, Rina Panigrahy, Manoj Prabhakaran, Amit Sahai, and Abhi Shelat. The smallest grammar problem. *Information Theory, IEEE Transactions on*, 51(7):2554–2576, 2005.
- [5] Francisco Claude and Gonzalo Navarro. Self-indexed grammar-based compression. *Fundamenta Informaticae*, 111(3):313–337, 2011.
- [6] Maxime Crochemore and Lucian Ilie. Computing longest previous factor in linear time and applications. *Information Processing Letters*, 106(2):75–80, 2008.
- [7] Travis Gagie. Large alphabets and incompressibility. *Information Processing Letters*, 99(6):246–251, 2006.
- [8] Rodrigo González and Gonzalo Navarro. Rank/select on dynamic compressed sequences and applications. *Theoretical Computer Science*, 410(43):4414–4422, 2009.
- [9] Sebastian Kreft and Gonzalo Navarro. Self-index based on LZ77 (Ph.D. thesis). *arXiv preprint arXiv:1112.4578*, 2011.
- [10] Sebastian Kreft and Gonzalo Navarro. Self-indexing based on LZ77. In *Combinatorial Pattern Matching*, pages 41–54. Springer, 2011.
- [11] Gonzalo Navarro and Yakov Nekrich. Optimal dynamic sequence representations. *SIAM Journal on Computing*, 43(5):1781–1806, 2014.

- [12] Enno Ohlebusch and Simon Gog. Lempel-Ziv factorization revisited. In *Combinatorial Pattern Matching*, pages 15–26. Springer, 2011.
- [13] Alberto Policriti and Nicola Prezza. Fast online lempel-ziv factorization in compressed space. In Costas Iliopoulos, Simon Puglisi, and Emine Yilmaz, editors, *String Processing and Information Retrieval*, volume 9309 of *Lecture Notes in Computer Science*, pages 13–20. Springer International Publishing, 2015.
- [14] Wojciech Rytter. Application of lempel–ziv factorization to the approximation of grammar-based compression. *Theoretical Computer Science*, 302(1):211–222, 2003.
- [15] Jouni Sirén et al. *Compressed full-text indexes for highly repetitive collections*. PhD thesis, 2012.
- [16] Jouni Sirén, Niko Välimäki, Veli Mäkinen, and Gonzalo Navarro. Run-length compressed indexes are superior for highly repetitive sequence collections. In *String Processing and Information Retrieval*, pages 164–175. Springer, 2009.
- [17] Tatiana Starikovskaya. Computing Lempel-Ziv factorization online. In *Mathematical Foundations of Computer Science 2012*, pages 789–799. Springer, 2012.
- [18] Yoshimasa Takabatake, Yasuo Tabei, and Hiroshi Sakamoto. Online self-indexed grammar compression. In *String Processing and Information Retrieval*, volume 9309 of *Lecture Notes in Computer Science*, pages 258–269. Springer International Publishing, 2015.
- [19] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on information theory*, 23(3):337–343, 1977.